Locally Abstract, Globally Concrete Semantics of Concurrent Programming Languages

Crystal Chang Din[2], Reiner Hähnle[1(✉)], Einar Broch Johnsen[2], Ka I Pun[2], and Silvia Lizeth Tapia Tarifa[2]

[1] Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany
haehnle@cs.tu-darmstadt.de
[2] Department of Informatics, University of Oslo, Oslo, Norway
{crystald,einarj,violet,sltarifa}@ifi.uio.no

Abstract. Language semantics that is formal and mathematically precise, is the essential prerequisite for the design of logics and calculi that permit automated reasoning about programs. The most popular approach to programming language semantics—small step operational semantics (SOS)—is not modular in the sense that it does not separate conceptual layers in the target language. SOS is also hard to relate formally to program logics and calculi. Minimalist semantic formalisms, such as automata, Petri nets, or π-calculus are inadequate for rich programming languages. We propose a new formal trace semantics for a concurrent, active objects language. It is designed with the explicit aim of being compatible with a sequent calculus for a program logic and has a strong model theoretic flavor. Our semantics separates sequential and object-local from concurrent computation: the former yields abstract traces which in a second stage are combined into global system behavior.

1 Introduction

Our goal in this paper is a new kind of trace semantics for concurrent OO programming languages with cooperative scheduling, no more, no less. It is designed with the explicit aim of being compatible with a sequent calculus for a program logic and has a strong model theoretic flavor. The semantics separates sequential and object-local from concurrent computations. This is achieved by keeping traces of local computations inside an abstract context. Only in a second stage these abstract traces are combined into global system behavior.

Motivation. Conspicuously, semantics is mostly absent in program logics, starting with the early work of Hoare [1] and Dijkstra [2] and extending to contemporary approaches, such as the program logic of the KeY system [3]. Generally, the rules of a calculus of a program logic are taken to *be* the (axiomatic) semantics and this is considered to be a major advantage: "axioms enable the language designer to express his general intentions quite simply and directly, without the mass of detail which usually accompanies algorithmic descriptions" [1, p. 583].

R.A. Schmidt and C. Nalon (Eds.): TABLEAUX 2017, LNAI 10501, pp. 22–43, 2017.
DOI: 10.1007/978-3-319-66902-1_2

Ample work on formalizing the semantics of programming languages has been performed. Many recent accounts are based on Plotkin's structural operational semantics (SOS).[1] In SOS the behavior of a program is formalized as transition rules that transform a given state of execution and a given program statement into a result state (which might be an abort state). The traces, i.e. the state sequences resulting from all possible sequences of rule applications from a given initial configuration, constitute a program's semantics.

Local SOS configurations contain runtime infrastructure, such as frame stacks, etc. The rules are schematic and need to cover all well-formed programs, in particular, the composition mechanisms of the target programming language, such as sequential composition, method calls, and synchronization. In essence, the SOS rules represent an interpreter.

Large fragments of industrial programming languages have been formalized in this style, including Java [5], C [6], but also the ABS language targeted here [7].

Yet, semantics based on transition systems tends not to be very modular. Often, there are dozens, if not hundreds of rules, and it is hard to judge the consequences if some of them are changed. It is also very difficult to relate SOS rules to modern verification calculi based on symbolic execution [3] or verification condition generation [8,9]. As noted above, the consequence is that the calculi of verification tools for mainstream languages generally lack a corresponding formal semantics relative to which soundness can be proven. Minimalist semantic formalisms, such as automata, Petri nets, or the π-calculus are frequently used in theoretical investigations, but they are inadequate for mainstream programming languages.

Contribution. A main contribution of our work is a new trace semantics for concurrent programs that exhibits a denotational, compositional flavor and strictly separates local from interleaving and parallel computations. It has a number of important advantages: 1. The semantics is concise: it has exactly one rule per statement. 2. Consequences of changes are local and easy to analyze. 3. It is denotational: Semantic objects (i.e. traces) directly result from application of semantic rules and not indirectly from their interpretation. 4. One can easily map semantic rules to rules of the program logic.

Below we introduce several new mechanisms into our trace semantics: Local computations cannot know their parallel execution context, therefore, we execute them in an *abstract* environment and we work with *continuations* to handle local suspension and blocking. This enables us to completely separate sequential from interleaving and parallel computations by means of *synchronization* events whose well-formedness is ensured when local, sequential behavior is composed into global, concurrent behavior. This can be viewed as a generalization of Brookes' action traces [10].

Specifically, our work targets the active object language ABS [7], but we expect that the principle is applicable to languages based on asynchronous communication in general and probably to other concurrent languages as well.

[1] The official citation is [4], but the approach goes back to the early 1980s.

$$
\begin{aligned}
\mathit{scope} &::= \{\mathit{block}\} \\
\mathit{block} &::= T\ \ell = \mathit{exp}; \mathit{block} \mid \mathit{stmt} \\
\mathit{stmt} &::= \mathit{block} \mid \texttt{skip} \mid \ell = \mathit{rhs} \mid \texttt{if}\ \mathit{bexp}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}\ \texttt{fi} \mid \texttt{return}\ \mathit{exp} \\
&\quad \mid \texttt{suspend} \mid \mathit{stmt}; \mathit{stmt} \mid \texttt{await}\ \mathit{gexp} \mid \texttt{while}\ \mathit{bexp}\ \{\ \mathit{stmt}\ \} \\
\mathit{rhs} &::= \mathit{exp} \mid \texttt{new}\ C\ (\bar{\ell}) \mid \mathit{cm} \\
\mathit{cm} &::= \ell!\mathit{mth}(\bar{\ell}) \mid \ell.\mathit{mth}(\bar{\ell}) \mid \ell.\texttt{get} \\
\mathit{gexp} &::= \mathit{bexp} \mid \ell? \\
\mathit{bexp} &::= \mathit{exp}\ \mathit{brel}\ \mathit{exp} \mid \mathrm{tt} \mid \mathrm{ff} \\
\mathit{brel} &::= == \mid < \mid > \mid \leq \mid \geq \\
\mathit{exp} &::= \mathit{exp} + \mathit{exp} \mid \mathit{exp} - \mathit{exp} \mid \mathit{exp} * \mathit{exp} \mid \mathit{exp}\ /\ \mathit{exp} \mid \ell \mid \texttt{this} \mid \texttt{destiny} \mid 0 \mid 1 \mid \cdots \\
\ell &::= \mathit{identifier} \mid \texttt{this}.\mathit{identifier}
\end{aligned}
$$

Fig. 1. ABS Program Syntax.

2 ABS: The Abstract Behavioral Specification Language

The Abstract Behavioral Specification language (ABS) [7] is an object-oriented modeling language for concurrent and distributed systems, which has been designed with a focus on analyzability. Its syntax and semantics are similar to Java to maximize usability. The grammar is given in Fig. 1. Expressions and imperative statements are standard. We slightly deviate from official ABS syntax and assume that local variables are introduced inside blocks before any other statement. This can be easily achieved by adding suitably scoped block statements. For simplicity, we only include integer and Boolean typed expressions, and we assume that all methods have a return value. The declaration of interfaces, classes, and methods is completely straightforward and omitted. We briefly discuss the main language features that are non-standard and the statements associated with them (for a full account, see [7]). The features considered here are explained by an example below.

Rigorous Encapsulation. Communication between different objects is only possible via method calls. The fields of an object are strictly private and inaccessible even to other instances of the same class and there are no static fields. There is no code inheritance and only interfaces constitute valid object types. This enforces the programming to interfaces discipline [11] and ensures that the heap of an object is only accessed by its own processes.

Asynchronous Communication with Futures. Asynchronous calls are dispatched with the statement `Fut<T> f = o!m(e)`, where method `m` is called on the object `o` with parameters `e`. Upon making this call, a *future* is bound to `f` and the caller continues its execution uninterrupted. A future is a handle to the called process and may be passed around; in particular, a process may refer to its own associated future using the keyword `destiny`. Upon process termination, its return value may be accessed via the associated future. To read a value from a future, the statement `T i = f.get;` is used.

Cooperative Scheduling. In ABS at most one process is active per object. Active processes cannot be preempted, but give up control when they suspend

```
1  interface IC { Int n(); }
2  class C implements IC {
3    Int i = 0;
4    Int m(Int x) {
5      this.i = x; return x;
6    }
7    Int n() {
8      Int y = 10;
9      Fut<Int> l1 = 0;
10     l1 = this ! m(3);
11     if y == 0 then y = this.m(1) else await l1? fi;
12     y = this.i+y;
13     return y;
14   }
15 }
16 { // Main block
17   Fut<Int> l = 0;
18   Int v = 0;
19   IC o = null;
20   o = new C();
21   l = o!n();
22   v = l.get;
23 }
```

Fig. 2. An example in ABS.

or terminate. Hence the ABS modeler has explicit control over interleaving. The active process suspends itself either by a `suspend` statement or by waiting for a guard. A guard can be a future—then the suspension statement has the form `await f?`; and the process may become active again once `f` has been resolved (i.e. its process terminated). Otherwise, a guard can be a side-effect-free Boolean expression—then the suspension statement has the form `await e`; and the process may become active again if `e` evaluates to true. If a future is accessed with `f.get` before it has been resolved, then the whole object blocks until `f` is resolved. When blocked, an object may still receive method calls, but it will not execute them.

The code between the start and the end of a method, as well as between suspension statements, can be reasoned about as if it were executed sequentially, because of cooperative scheduling: the active process is guaranteed to have exclusive access to the local heap of its object.

Given in Fig. 2 is an ABS example covering most ABS syntax categories. It declares an interface IC with a single public method, implemented in class C and called in the program's Main block. After declaring a future l and variables v, o, a new C object is created and stored in o. As o is new, it runs on a separate processor from Main. The asynchronous call to `n` returns a future that is stored in l and the following `get` is immediately executed. If the call to `n` is not yet completed, the main process blocks until it is. As there is no other process waiting on o, the call to `n` will be scheduled provided the scheduler is fair.

Local variable declarations of y and l_1 are followed by an asynchronous self call to `m`. This cannot start yet, because the execution of `n` continues with the conditional, where the second branch is taken. At this point, the execution of `n` suspends, and the execution of `m` can commence. In `m` the field `i` is set to the call parameter, hence `this.i` == `3` holds when `n` resumes. Consequently, `n` returns the value 13, which will also be assigned to v.

3 Abstract Traces

Our goal is a denotational trace semantics for ABS that, given a piece of ABS code, yields all possible traces for any local context, where a trace is a finite or infinite sequence of computation states. By *local* we mean an arbitrary initial state, an object `this` (including the heap), and a future `destiny`. Since ABS is non-deterministic, it is clear that we require a *collecting semantics*, i.e. a set of traces. More importantly, we want the semantics to be *local* in the sense that the evaluation of a given piece of ABS code is *independent* of the evaluation of other pieces. In a second stage, the local evaluations will be composed into well-formed traces that constitute the global system behavior. At first glance, this seems completely impossible. Look at the following code snippet (closely related to the example in Fig. 2):

$$v = f.\texttt{get};\ \texttt{if}\ (v == 0)\ \texttt{then}\ o.m()\ \texttt{else await}\ f'? \tag{1}$$

Behavior of this code depends, for example, on whether the futures f and f' have been resolved or not. Moreover, we need to know on which object o the method m is called. The branch taken in the conditional depends on the value of v, which is in turn the result of a previous asynchronous computation. So how can we achieve locality? The central idea is to abstract away from the unknowns during the evaluation of local statements:

1. As we cannot know which execution branch is taken, we simply generate traces for all of them. Hence, the evaluation of a statement yields a *set* of traces. This is standard in cumulative semantics [12].
2. Likewise, we don't know the values of parameters and attributes and, in particular, of the initial state, so the semantic evaluation will be *symbolic*.
3. Likewise, as we don't know the identity of `this` object, nor that of the future `destiny`, evaluation is *parametric* in them.

Yielding from these ideas is a formalization where we define, for any ABS statement s, a valuation function $\mathrm{val}_\sigma^{O,F}(s)$ that returns the *set* of possible traces resulting from s when started in (a symbolic) initial state σ on object O with `destiny` F. Throughout the paper we assume a fixed domain D of semantic values and drop it from the valuation function to improve readability. We further assume that all expressions are well-typed and that their evaluation on D is fixed in a standard manner (e.g., + is addition on the integers, etc.). As indicated above, we need to be able to evaluate the semantics in symbolic states:

Definition 1 (Memory location, symbolic/concrete state). Consider the set of all objects $\mathcal{O}$, all futures $\mathcal{F}$, the set of local variables and parameters $\mathcal{V}$, the set of attributes (fields) $\mathcal{A}$, and let $\mathcal{C}$ be an infinite supply of symbolic constants. Denote by $\mathcal{L} = \{O.a \mid O \in \mathcal{O}, a \in \mathcal{A}\} \cup \mathcal{V} \cup \mathcal{C} \cup \mathcal{F}$ the *set of all memory locations* and by $\mathsf{Exp}(\mathcal{L})$ the ABS *expressions* over $\ell \in \mathcal{L}$. A *symbolic state* σ is a function $\sigma : \mathcal{L} \to \mathsf{Exp}(\mathcal{L})$. Without loss of generality we assume that $\sigma(\ell)$ is fully evaluated whenever $\sigma(\ell)$ contains no symbol from $\mathcal{L}$, i.e. $\sigma(\ell) \in D$. A state is called a *concrete state* if its range is in D and a trace is called a *concrete trace* if all its states are concrete.

Our evaluation of (1) starts in a state where $\sigma(v) = v_0$ and $v_0 \in \mathcal{C}$ is a fresh constant. Semantic evaluation of a statement returns a set of traces τ over symbolic states. Allowing symbolic constants in states amounts to a restricted form of *symbolic execution*. In order to decide which traces are feasible and which are not, it is necessary to record *path conditions* whenever a trace splits, such as for the conditional statement in the example above. Therefore, we work with *conditioned, symbolic traces* of the form $pc \triangleright \tau$, where pc is a path condition that must be satisfiable for the trace τ to be feasible. Path conditions are sets of quantifier-free Boolean expressions over $\mathcal{L}$. For (1) we would obtain path conditions of the form $\{v_0 = 0\}$ and $\{v_0 \neq 0\}$.

Now, during the semantic evaluation, we ensure that the value $\sigma(\ell)$ of each symbol ℓ in the path condition and in the symbolic states of a trace τ is correctly maintained relative to the initial state in τ [13]. Consequently, the symbolic traces can be easily concretized:

Definition 2 (Concretization of symbolic trace). Conditioned, symbolic traces are denoted by $pc \triangleright \tau$ and concrete states by σ. The *concretization* τ_σ of τ is obtained by replacing each σ' in τ with $\sigma \circ \sigma'$ and pc with $\sigma(pc)$. We can assume that path conditions are automatically evaluated, so that $\sigma(pc)$ is either true or false (where empty path conditions are considered to be true).

Notation for Traces. Represent symbolic traces in the sequel by variables τ, ω, where τ is typically finite and ω infinite. We use sh for concrete traces, i.e. traces in the usual sense (the letters stand for "shining trace"). Path conditions, if present, are explicitly given, but we identify $\mathsf{true} \triangleright sh$ with sh. The constructors for traces (symbolic or concrete) are as follows: The empty trace is written ε. Given a (possibly empty) trace τ, we extend it with a single state σ by writing $\tau \curvearrowright \sigma$. A singleton trace consisting of the state σ is written $\langle\sigma\rangle = \varepsilon \curvearrowright \sigma$. Concatenation of two traces τ, ω is written as $\tau \cdot \omega$ and only defined when τ is finite. The final state of a non-empty, finite trace τ is obtained as $\mathrm{last}(\tau)$.

Extend sequential composition of programs to traces as follows: Assume that τ is a trace of statement r and ω a trace of s. To obtain the trace corresponding to the sequential composition of r and s, the last state of τ and the first state of ω must be identical, but the resulting trace should not contain a doubled state. Hence, we define the *semantic chop* $\underline{**}$ (inspired by [14]):

$$(pc_\tau \triangleright \tau) \underline{**} (pc_\omega \triangleright \omega) = \begin{cases} pc_\tau \triangleright \tau & \text{if } \tau \text{ is infinite or } \tau = \tau' \cdot \mathit{starve}(O) \\ (pc_\tau \cup pc_\omega) \triangleright \tau \cdot \omega' & \text{if } \mathrm{last}(\tau) = \sigma,\ \omega = \langle\sigma\rangle \cdot \omega' \end{cases} \tag{2}$$

This definition takes into account a possibly non-terminating first trace, either because it is infinite or it represents a starving process ($\mathit{starve}(O)$ is a starvation marker defined below). The definition can be specialized to traces without path conditions in the obvious way: $\tau \underline{**} \omega$ is $(\emptyset \triangleright \tau) \underline{**} (\emptyset \triangleright \omega)$.

4 The Local Semantics of ABS Programs

Execution of ABS will now be formalized using denotational and compositional local semantics; the main challenge here is the suspension of local control flow. To meet this challenge, we introduce continuations. We first consider statements which do not need continuations, before we discuss the continuation mechanism.

4.1 Statements Without Continuations

Syntactic structure of programs guides the definition of the valuation function $\mathrm{val}_\sigma^{O,F}(s)$. As explained above, given a symbolic state σ, $\mathrm{val}_\sigma^{O,F}(s)$ yields the set of all possible symbolic traces when s is executed from the initial state σ on object O with destiny F. We explain it case by case, beginning with the valuation of scopes that start with local variable declarations; here, bs may contain further variable declarations, followed by a statement, see Fig. 1:

$$\begin{aligned} \mathrm{val}_\sigma^{O,F}(\{T\,\ell = e; bs\}) \ = \{&pc \triangleright \langle\sigma\rangle \cdot \omega \mid \sigma' = \sigma[\ell' \mapsto \mathrm{val}_\sigma^{O,F}(e)], \\ &pc \triangleright \omega \in \mathrm{val}_{\sigma'}^{O,F}(\{bs[\ell'/\ell]\}), \mathit{isFresh}(\ell')\}. \end{aligned} \tag{3}$$

Evading name clashes between variable names is achieved by replacing ℓ with a fresh name ℓ' throughout the scope and evaluate the resulting scope in the state σ', where ℓ' has been initialized with the evaluation of e. The evaluation function for side effect-free expressions $\mathrm{val}_\sigma^{O,F} : \mathsf{Exp}(\mathcal{L}) \to \mathsf{Exp}(\mathcal{L})$ is completely standard, except $\mathrm{val}_\sigma^{O,F}(\texttt{this}) = O$ and $\mathrm{val}_\sigma^{O,F}(\texttt{destiny}) = F$. The traces of the form $pc \triangleright \omega$ resulting from evaluation of the renamed scope start with σ', so we need to prepend σ. The next rule evaluates scopes without leading local variable declarations. These become simply statements after stripping the delimiting braces:

$$\mathrm{val}_\sigma^{O,F}(\{s\}) = \mathrm{val}_\sigma^{O,F}(s)$$

Meanwhile, the `skip` statement yields a trace of length one with empty path condition:

$$\mathrm{val}_\sigma^{O,F}(\texttt{skip}) = \{\emptyset \triangleright \langle\sigma\rangle\} \tag{4}$$

Assignment of expressions results in a single trace of length two: from the initial state σ to the state where ℓ has been updated with the value of e:

$$\mathrm{val}_\sigma^{O,F}(\ell = e) = \{\emptyset \triangleright \langle\sigma\rangle \frown \sigma[\ell \mapsto \mathrm{val}_\sigma^{O,F}(e)]\} \tag{5}$$

Notation for Events in Traces. To model concurrency in our semantics we use *event markers* in traces. For example, an object must have been created before its methods can be called. With event markers it is easy to ensure such properties via well-formedness conditions over events. For example, each invocation event on an object o in a given trace must be preceded by a creation event for o. Let $ev(\overline{v})$ be an event marker with arguments $\overline{v}$. To insert $ev(\overline{v})$ into a trace that continues with σ, we define an *event trace* $ev_\sigma(\overline{v})$ of length three as follows:

$$ev_\sigma(\overline{v}) = \langle\sigma\rangle \curvearrowright ev(\overline{v}) \curvearrowright \sigma.$$

This notation has the advantage that it is "choppable" with preceding or trailing traces and it ensures that no trace begins or ends with an event marker. Evaluation of the assignment of new objects is now straightforward:

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\ell = \texttt{new}\ C\,(\overline{e})) = \{pc \triangleright newEv_\sigma(O, o, \mathrm{val}_\sigma^{O,F}(\overline{e})) \cdot \tau \mid \\ isFresh(o),\ class(o) = C,\ \sigma' = C.\epsilon(o) \circ \sigma, \\ pc \triangleright \tau \in \mathrm{val}_{\sigma'}^{O,F}(\ell = o)\}\end{aligned} \tag{6}$$

Insertion of the event marker *newEv* happens at the initial state σ. This creation event is attached to the current object O and represents the creation of a new object o whose class must be C. To initialize o, define an *abstract* initial state $C.\epsilon(o)$ such that $(C.\epsilon(o))(a) = a_0$ for each attribute a of C, where a_0 is a fresh symbol of the same type as a, and let σ' extend σ with those initial assignments. The assignment of the newly created object can then be evaluated by rule (5). The construction is deterministic and results in a single trace of length five. *Asynchronous* method calls follow a similar schema:

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\ell = e'!m(\overline{e})) = \{pc \triangleright invEv_\sigma(O, \mathrm{val}_\sigma^{O,F}(e'), f, m, \mathrm{val}_\sigma^{O,F}(\overline{e}))\ \underline{**}\ \tau \mid \\ isFresh(f),\ method(f) = m,\ pc \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = f)\}\end{aligned} \tag{7}$$

An event marker *invEv* from the caller O to the evaluated callee $\mathrm{val}_\sigma^{O,F}(e')$ is inserted at state σ. The invocation event is associated with a fresh future f, the name m of the called method, and the evaluated call arguments $\mathrm{val}_\sigma^{O,F}(\overline{e})$. The call does not change the state and does not suspend, so we can assign the future to ℓ and proceed. Here the local semantics shows its strength: the called method is evaluated separately; only later will we take care of synchronization. *Synchronous* method calls are not handled using futures, but by inlining:

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\ell = e'.m(\overline{e})) = \{pc \triangleright \langle\sigma\rangle \cdot \omega \mid O' = \mathrm{val}_\sigma^{O,F}(e'), \\ lookup(m, class(O')) = T\ m(\overline{T}\ \overline{\ell'})\{s; \texttt{return}\ e\}, \\ pc \triangleright \omega \in \mathrm{val}_{\sigma'}^{O',F}(\{\overline{T}\ \overline{\ell'} = \mathrm{val}_\sigma^{O,F}(\overline{e}); s; \ell'' = e\}; \ell = \ell''), \\ \sigma' = \sigma[\ell'' \mapsto v_0],\ isFresh(\ell'', v_0)\}\end{aligned} \tag{8}$$

We get all traces $pc \triangleright \omega$ of the called method, whose implementation is obtained from a class table lookup of the callee O'. Observe that even when O' is a symbolic value, we can still determine its type statically. We initialize

the formal parameters with the call parameters and put the resulting code into a scope. Consequently, the formal parameters are treated as local variables and automatically renamed. A fresh variable ℓ'' will hold the return value and assign it to ℓ. This causes the only slight complication, because ℓ'' needs to be initialized with a fresh value in the state σ' from where the inlined code is executed. Here, the predicate *isFresh* expresses that one or more variable names are (globally) fresh.

For the evaluation of conditionals, we take the union of the sets of behaviors of the branches and add appropriate path conditions for each branch:

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ fi}) = \\ \{\{\mathrm{val}_\sigma^{O,F}(e) = \mathrm{tt}\} \cup pc_1 \triangleright \omega_1 \mid pc_1 \triangleright \omega_1 \in \mathrm{val}_\sigma^{O,F}(s_1)\} \ \cup (9)\\ \{\{\mathrm{val}_\sigma^{O,F}(e) = \mathrm{ff}\} \cup pc_2 \triangleright \omega_2 \mid pc_2 \triangleright \omega_2 \in \mathrm{val}_\sigma^{O,F}(s_2)\}\end{aligned}$$

We discuss two further cases for sequential statements, before we turn to rules with continuations. The `return` statement emits an event marker *compEv* for completion, given the current object and future to contain the returned value.

$$\mathrm{val}_\sigma^{O,F}(\texttt{return } e) = \{\emptyset \triangleright \mathit{compEv}_\sigma(O, F, \mathrm{val}_\sigma^{O,F}(e))\} \tag{10}$$

Finally, consider execution of a method m declared in class C, running on object O with future F: it starts with an event marker *invREv* representing the reaction to an asynchronous invocation from an unknown caller O' and with unknown argument values $\overline{v_0}$. This is followed by any of the possible traces for the method's implementation. The formal parameters are handled as local variables initialized with $\overline{v_0}$ that are put into a scope over the method body s.

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(C.m) = \{pc \triangleright \mathit{invREv}_\sigma(O', O, F, m, \overline{v_0}) \underline{**} \omega \mid \\ pc \triangleright \omega \in \mathrm{val}_\sigma^{O,F}(\{\overline{T}\ \overline{\ell'} = \overline{v_0}; s\}), \qquad (11)\\ \mathit{lookup}(m, C) = T\ m(\overline{T}\ \overline{\ell'})\{s\},\ \mathit{isFresh}(O', \overline{v_0})\}\end{aligned}$$

4.2 Statements with Continuations

All remaining statements may involve suspension, the intermittent scheduling of other processes, and resumed execution of the suspended statement. Two problems must be addressed: first, we cannot know the computation state upon resumption. Second, when we later combine local into global traces, we require interleaving points. Both are addressed by continuations. We start with the simplest case, the unconditional `suspend` statement:

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\texttt{suspend}) = \{\emptyset \triangleright \mathit{relEv}_\sigma(O) \cdot \mathit{starve}(O)\} \cup \\ \{\emptyset \triangleright \mathit{relEv}_\sigma(O) \cdot \mathit{relCont}(O, F, \texttt{skip})\} \qquad (12)\end{aligned}$$

Release of control of the currently executing process is captured by an event marker *relEv*. It has the current object as argument to identify which object

was released, when global system behavior is composed. Execution after suspension has two cases. First, the current object may suffer from starvation and never regain control. This situation is captured by the marker *starve*(O), which can only occur as the final element of a trace. Second, control is regained but we do not know what happened "in between" while other processes were executing on O. Without knowing the state in which execution will continue, it is not meaningful to evaluate the rest of the process. This is only achieved later, when we combine the local, sequential evaluation into the global one. Technically, we address this problem by ending the trace with a *continuation marker relCont*, which captures the return of control after suspension. The arguments of a continuation marker are the currently executing object, the future associated with the computation, and the code to be executed after control is regained. For unconditional suspension this is just a `skip` statement. The `get` statement, which retrieves the value of an asynchronous computation, can also introduce a continuation marker. There are two branches:

$$\begin{aligned}\mathrm{val}_\sigma^{O,F}(\ell = \ell'.\texttt{get}) = \{pc \triangleright \mathit{compREv}_\sigma(O, \mathrm{val}_\sigma^{O,F}(\ell'), v_0) \mathbin{\underline{**}} \tau \mid \\ \mathit{isFresh}(v_0),\ pc \triangleright \tau \in \mathrm{val}_\sigma^{O,F}(\ell = v_0)\} \cup \\ \{\emptyset \triangleright \mathit{blkEv}_\sigma(O, \mathrm{val}_\sigma^{O,F}(\ell')) \cdot \mathit{blkCont}(O, F, \ell = \ell'.\texttt{get})\}\end{aligned} \tag{13}$$

In the first branch, the future bound to ℓ' has been resolved. We introduce an event marker *compREv* to capture a completion reaction in the current object O. The actual result is unknown at this point, so we assign a fresh, symbolic value to ℓ, to represent it. This causes no problem in a symbolic setting, and the value will be later resolved during composition of global behavior. In the second branch, the process is scheduled to retrieve the value of an unresolved future. The process is blocked until the future is resolved, i.e. local control is not released. This is captured by an event marker *blkEv*, associated to the current object O and the future. Similar to unconditional suspension, we do not know how the state evolves while the process is blocked and put a continuation marker *blkCont* at the end of the local trace to enable the correct composition of global traces.

In the sequel we use the convention that traces denoted with τ do *not* contain any continuation marker. To fully understand the continuation mechanism it is useful to look at last branches of the evaluation of sequential composition:

$$\begin{aligned}&\mathrm{val}_\sigma^{O,F}(r; s) = \\ &\quad \{(pc_r \triangleright \tau_r) \mathbin{\underline{**}} (pc_s \triangleright \omega_s) \mid pc_r \triangleright \tau_r \in \mathrm{val}_\sigma^{O,F}(r), pc_s \triangleright \omega_s \in \mathrm{val}_{\sigma'}^{O,F}(s), \\ &\qquad \text{where } \sigma' = \mathrm{last}(\tau_r) \text{ if } \tau_r \text{ is } \mathsf{finite}, \text{ arbitrary otherwise}\} \cup \\ &\quad \{pc_r \triangleright \tau_r \cdot \mathit{relCont}(O, F, r'; s) \mid pc_r \triangleright \tau_r \cdot \mathit{relCont}(O, F, r') \in \mathrm{val}_\sigma^{O,F}(r)\} \cup \\ &\quad \{pc_r \triangleright \tau_r \cdot \mathit{blkCont}(O, F, r'; s) \mid pc_r \triangleright \tau_r \cdot \mathit{blkCont}(O, F, r') \in \mathrm{val}_\sigma^{O,F}(r)\}\end{aligned} \tag{14}$$

Let us evaluate $\texttt{suspend}; s$. By (12) we obtain a set of traces for `suspend` that end with $\mathit{relCont}(O, F, \texttt{skip})$. By (14) the evaluation of $\texttt{suspend}; s$ is a set of traces that end with $\mathit{relCont}(O, F, \texttt{skip}; s)$. All evaluations accumulate the remaining commands in this way, so the top-level continuation contains all code

remaining to be executed. For $\ell = \ell'.\texttt{get}$ we similarly obtain from (13) a set of traces that end with $blkCont(O, F, \ell = \ell'.\texttt{get})$, which can be sequenced with s to $blkCont(O, F, \ell = \ell'.\texttt{get}; s)$. This shows that the `get` statement is re-evaluated before the process can proceed with the remaining statements. The first branch of sequential composition covers the sequential case: r runs without suspension or blocking. This includes the case when r starves or does not terminate (2).

$$\begin{aligned} \mathrm{val}_\sigma^{O,F}(\texttt{await } \ell?) = {} & \{\emptyset \triangleright compREv_\sigma(O, \mathrm{val}_\sigma^{O,F}(\ell), v_0) \mid isFresh(v_0)\} \cup \\ & \{\emptyset \triangleright relEv_\sigma(O, \mathrm{val}_\sigma^{O,F}(\ell)) \cdot starve(O)\} \ \cup \\ & \{\emptyset \triangleright relEv_\sigma(O, \mathrm{val}_\sigma^{O,F}(\ell)) \cdot relCont(O, F, \texttt{await } \ell?)\} \end{aligned} \tag{15}$$

Awaiting a future is similar to suspension (12). Its second and third branch are almost the same, but in addition to the executing object we need to record the identity of the future ℓ in the release event marker *relEv*, to ensure well-formedness of traces in the global semantics. In the third branch, the `await` statement is re-evaluated in the continuation. This models `await` as a loop that repeatedly suspends and checks whether the future is available. The latter is treated in the first branch: to capture the completion reaction, we insert an event marker *compREv*, for the future in ℓ and record its value. This value is yet unknown and set to a symbolic term v_0. But how do we know that the future was actually completed in the first branch? In the local evaluation we cannot know and we might create ill-formed traces at this point; such traces will be removed later when we compose the global behavior. Await on fields can be defined as: $\mathrm{val}_\sigma^{O,F}(\texttt{await } e) = \mathrm{val}_\sigma^{O,F}(\texttt{if } e \texttt{ then skip else suspend; await } e \texttt{ fi})$. An inductive version of loop evaluation can be defined similarly, using the rules above: $\mathrm{val}_\sigma^{O,F}(\texttt{while } e \ \{s\}) = \mathrm{val}_\sigma^{O,F}(\texttt{if } e \texttt{ then } s\texttt{; while } e \ \{s\} \texttt{ else skip fi})$.

4.3 Local Traces by Example

Figure 3 summarizes the local, abstract traces of the example from Fig. 2. Denote the empty state by ϵ and a state σ such that $\sigma(\ell) = v$ by $[\ell \mapsto v]$; i.e. states are unnamed and only relevant parts of their domain are listed. The evaluation of method `m` in $\mathrm{val}_{\mathsf{C}.\epsilon(O)}^{O,F}(\mathsf{C.m})$ gives a set with one symbolic trace (note that parameter `x` was renamed to x'), while the evaluation of method `n` in $\mathrm{val}_{\mathsf{C}.\epsilon(O)}^{O,F}(\mathsf{C.n})$ gives a set with four symbolic traces due to the `if−then−else` and `await` statements.

We discuss the traces for method `n` in detail. The first trace occurs when the condition of the `if−then−else` statement is true, and we proceed with the execution of the synchronous call to method `m`, which has no release points. The next three cases occur when the condition of the `if−then−else` statement is false, and we evaluate the `await` statement. In the second trace the future has been resolved; we do not have a release point and the trace contains a completion event. The third trace occurs when the future is unresolved. After release, the process never regains control, i.e. the scheduler is unfair, so the trace ends with a starvation marker. The fourth trace happens when the process regains control with a release continuation $relCont(O, F, \ldots)$. The evaluation of

$$\mathrm{val}^{O,F}_{\mathsf{C}.\epsilon(O)}(\mathsf{C.m}) \;=$$
$$\{\quad \emptyset \quad \blacktriangleright\langle [O.i \mapsto v_i]\rangle \curvearrowright invREv(O', O, F, \mathsf{m}, v_0) \curvearrowright \cdots \curvearrowright [O.i \mapsto v_i, x' \mapsto v_0] \quad\}$$

$$\mathrm{val}^{O,F}_{\mathsf{C}.\epsilon(O)}(\mathsf{C.n}) \;=$$
$$\begin{array}{l}\{\{(10=0)\} \triangleright\langle [O.i \mapsto v_j]\rangle \curvearrowright invREv(O', O, F, \mathsf{n}, \_) \curvearrowright \ldots\\ \quad \{(10\neq 0)\} \triangleright\langle [O.i \mapsto v_j]\rangle \curvearrowright invREv(O', O, F, \mathsf{n}, \_) \curvearrowright \cdots \curvearrowright compEv(O, F, v_j+10) \curvearrowright\\ \qquad [O.i \mapsto v_j, y' \mapsto v_j+10, l'_1 \mapsto f_1],\\ \quad \{(10\neq 0)\} \triangleright\langle [O.i \mapsto v_j]\rangle \curvearrowright invREv(O', O, F, \mathsf{n}, \_) \curvearrowright \ldots starve(O),\\ \quad \{(10\neq 0)\} \blacktriangleright\langle [O.i \mapsto v_j]\rangle \curvearrowright \cdots \curvearrowright relEv(O, f_1) \curvearrowright [O.i \mapsto v_j, y' \mapsto 10, l'_1 \mapsto f_1]\\ \qquad \cdot relCont(O, F, \texttt{await}\ l'_1?; y' = \texttt{this}.i + y'; \texttt{return}\ y';) \quad\}\end{array}$$

$$\begin{array}{l}\mathrm{val}^{O,F}_{[O.i\mapsto v_{j'}, y'\mapsto v_{y'}, l'_1\mapsto v_2]}(O, F, \texttt{await} l'_1?; y' = \texttt{this}.i + y'; \texttt{return}\ y';) \;=\\ \{\{(v_{y'}\neq 0)\} \blacktriangleright\langle [O.i \mapsto v_{j'}, y' \mapsto v_{y'}, l'_1 \mapsto v_2]\rangle \curvearrowright \cdots \curvearrowright compEv(O, v_2, v'_j + v'_y) \curvearrowright\\ \qquad [O.i \mapsto v_{j'}, y' \mapsto v'_j + v'_y, l'_1 \mapsto v_2],\\ \quad \{(v_{y'}\neq 0)\} \triangleright\langle [O.i \mapsto v_{j'}, y' \mapsto v_{y'}, l'_1 \mapsto v_2]\rangle \curvearrowright \ldots starve(O),\\ \quad \{(v_{y'}\neq 0)\} \triangleright\langle [O.i \mapsto v_{j'}, y' \mapsto v_{y'}, l'_1 \mapsto v_2]\rangle \curvearrowright \ldots\\ \qquad relCont(O, F, \texttt{await}\ l'_1?; y' = \texttt{this}.i + y'; \texttt{return}\ y';) \quad\}\end{array}$$

$$\begin{array}{l}\mathrm{val}^{\mathsf{Main},\, f_0}_{\epsilon}(\{\mathtt{Fut}\langle \mathsf{Unit}\rangle\ l = \mathsf{null};\ \mathsf{Int}\ v = 0;\ \mathsf{IC}\ o = \mathsf{null}; o = \texttt{new}\ \mathsf{C}(); l = o!\mathsf{n}(); v = l.\texttt{get};\}) =\\ \{\quad \emptyset \quad \triangleright\varepsilon \curvearrowright [l' \mapsto null] \curvearrowright \cdots \curvearrowright invEv(\mathsf{Main}, o'', f, \mathsf{n}, \_) \curvearrowright \cdots \curvearrowright\\ \qquad compREv(\mathsf{Main}, f, v_1) \curvearrowright \cdots \curvearrowright [l' \mapsto f, v' \mapsto v_1, o' \mapsto o'', o''.i \mapsto 0],\\ \qquad \emptyset \quad \blacktriangleright\varepsilon \curvearrowright \cdots \curvearrowright invEv(\mathsf{Main}, o'', f, \mathsf{n}, \_) \curvearrowright \cdots \curvearrowright blockEv(\mathsf{Main}, f) \curvearrowright\\ \qquad [l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 0] \cdot blkCont(\mathsf{Main}, f_o, v = l.\texttt{get};) \quad\}\end{array}$$

$$\begin{array}{l}\mathrm{val}^{\mathsf{Main},\, f_0}_{[l'\mapsto v_3, v'\mapsto v_4, o'\mapsto o'', o''.i\mapsto v_5]}(\{v = l.\texttt{get};\}) =\\ \{\quad \emptyset \quad \triangleright\langle [l' \mapsto v_3, v' \mapsto v_4, o' \mapsto o'', o''.i \mapsto v_5]\rangle \curvearrowright compREv(\mathsf{Main}, v_3, v_6) \curvearrowright \cdots \curvearrowright\\ \qquad [l' \mapsto v_3, v' \mapsto v_6, o' \mapsto o'', o''.i \mapsto v_7],\\ \qquad \emptyset \quad \blacktriangleright\langle [l' \mapsto v_3, v' \mapsto v_4, o' \mapsto o'', o''.i \mapsto v_5]\rangle \curvearrowright blockEv(\mathsf{Main}, f) \curvearrowright\\ \qquad [l' \mapsto v_3, v' \mapsto v_4, o' \mapsto o'', o''.i \mapsto v_5] \cdot blkCont(\mathsf{Main}, f_o, v = l.\texttt{get};) \quad\}\end{array}$$

Fig. 3. Examples of local traces. The filled triangle ▶ identifies those traces that become part of the global trace in Sect. 5.4.

$\mathrm{val}^{O,F}_{[O.i\mapsto v_{j'}, y'\mapsto v_{y'}, l'_1\mapsto v_2]}(O, F, \texttt{await}\ l'_1?; \ldots)$ results in three traces due to the `await` statement, following a similar pattern as the three last traces of `n`.

The evaluation of the main block in $\mathrm{val}^{\mathsf{Main},\, f_0}_{\epsilon}(\{\mathtt{Fut}\langle \mathsf{Unit}\rangle\ l = \mathsf{null}; \ldots\})$ produces two traces due to the `get` statement. In the first trace, the future is resolved and the main block finishes execution. In the second trace, the future is not resolved and we get a blocking continuation $blkCont(\mathsf{Main}, f_0, \ldots)$, marking the object as blocked until the future is resolved. The evaluation of $\mathrm{val}^{\mathsf{Main},\, f_0}_{[l'\mapsto v_3, v'\mapsto v_4, o'\mapsto o'', o''.i\mapsto v_5]}(\{v = l.\texttt{get};\})$ again produces two traces, similar as above. Note that some of these symbolic traces will never result in a concrete trace of the global system, because they will be eliminated by well-formedness requirements at the global level or due to inconsistencies in their path conditions.

5 The Global Semantics of ABS Programs

The local semantics of ABS yields for each object $O \in \mathcal{O}$, future $F \in \mathcal{F}$, and statement s a set of conditioned, symbolic traces $\omega \in \mathrm{val}^{O,F}_{\sigma}(s)$ that describe all possible behaviors of s when started in state σ. We now construct, for a given ABS program P, a set of *concrete* traces sh without path conditions describing

the possible *global* behaviors of the system; i.e. these traces consist of *global states* that fix the value of each variable and each attribute of each object.

5.1 From Locally Abstract to Globally Concrete Behavior

An ABS execution starts from an executable main block $\{\overline{T}\ \overline{\ell} = \overline{v}; s\}$ of an ABS program P in a concrete global state ϵ, where each attribute and variable is initialized with the concrete default value of its type. We assume that the main block is executed on object Main that is associated with a future f_0.[2] Our semantics works as follows: we start to evaluate the main block by picking a trace from $\mathcal{M} = \mathrm{val}_{\epsilon}^{\mathsf{Main},\, f_0}(\{\overline{T}\ \overline{\ell} = \overline{v}; s\})$. As long as we don't suspend execution, this results in a *concrete* trace with path condition either true or false, because s is executable and fresh values are only introduced upon suspension. We only produce traces with feasible path condition, i.e. true, which can be discarded. Hence, the result is an initial trace sh of P.

Two technical issues need to be addressed. The first is *suspension*, given by continuation markers in the symbolic traces. When we encounter a continuation in a symbolic trace, other traces on O and on other objects should have a chance to be inserted in the global trace. Where do these other traces come from? In addition to $\mathcal{M}$ we have the following symbolic traces at our disposal:

$$\mathcal{G} = \{\mathrm{val}_{\mathsf{C}.\epsilon(O)}^{O,\, F}(C.m) \mid \mathit{class}(O) = \mathsf{C},\ m \in \mathit{mtd}(\mathsf{C}),\ O \in \mathcal{O},\ F \in \mathcal{F},\ \mathsf{C} \in P\} \quad (16)$$

Symbolic traces represent possible executions on different objects, started in abstract states (6) for all objects, all futures, and for each method of each class in P. Assume that our initial concrete trace sh contains an invocation event $\mathit{invEv}_{\sigma}(O, v, F, m, \overline{v})$ and that we have suspended trace generation. Note that v and $\overline{v}$ are *concrete* values in D. We select a symbolic trace in $\mathrm{val}_{\mathsf{C}.\epsilon(O)}^{O,\, F}(C.m) = \Omega \in \mathcal{G}$ and instantiate it with v and $\overline{v}$. In addition, we start it with the *concrete* state $\mathrm{last}(sh)$ instead of $\mathsf{C}.\epsilon(O)$. Well-formedness conditions over event sequences will ensure that only valid ABS traces can be generated in this way.

We formalize this idea in *global trace composition rules*. These define a relation $\rightarrow$ that takes a concrete initial trace sh and a queue q of sets of local symbolic traces, extends sh by concretizing one such trace, and modifies q accordingly. Exhaustive, non-deterministic application of these rules yields one of the possible global system traces of P. The initial state of that global execution is given by:

$$\varepsilon, \{\mathcal{M}\} \cup \mathcal{G}$$

The second technical issue is *non-termination*. Starvation is straightforward, due to the event markers in the local semantics. When we encounter a starving object, we can simply discard all traces associated with it, and let other objects continue execution. However, non-terminating statements, such as loops or synchronous recursive calls, contain neither continuation nor starvation markers.

[2] This future is never retrieved by any completion reaction event and can be thought of as the client who started P's execution.

To produce a global system trace, we need to interrupt the generation of such infinite traces "from time to time" so that other objects in the global system (except the diverging object) can proceed. Our solution to this problem is to let the generation of concrete traces be preempted after some finite number of steps, but we need to exclude arbitrary interleaving of traces, which is not permitted by the cooperative concurrency model of ABS. Technically, this can be done by means of *interleaving events* and *interleaving reaction events* that contain enough information to exclude unwanted traces. It has previously been shown that local scheduling information of this kind is needed to obtain a complete proof system for cooperative concurrency [15].

5.2 The Rules of the Global Semantics

The correct global composition of traces is governed by *events* over futures and objects, which are related by a *well-formedness predicate* over global traces. While the correct interaction with futures depends on the communication events introduced in Sect. 4, the interleaving of different executions is captured by two kinds of scheduling events, related to internal and external interleaving of execution. Internal interleaving reflects the cooperative concurrency of ABS objects, with `suspend` and `await`. We let the event $schEv_\sigma(O)$ express that O has scheduled a process in state σ. External interleaving reflects how the execution in different objects may be interleaved in the global trace. This is captured in the semantics by a pair of interleaving events, $ilEv_\sigma(O)$ and $ilREv_\sigma(O)$, expressing that object O permits the execution of other objects to be observed in state σ and that object O continues its execution after such an observation, respectively.

We define the execution relation $\rightarrow$ for global execution by five composition rules. The first rule captures *external interleaving* in the global trace by preempting the local execution:

$$\frac{\begin{array}{c} pc \triangleright \tau \cdot \omega \in \Omega \qquad \Omega \in q \qquad object(\Omega) = O \qquad \mathrm{last}(sh) = \sigma \\ \tau \neq \varepsilon \qquad \omega \notin \{\varepsilon, blkCont(O,\_,\_), relCont(O,\_,\_), starve(O)\} \\ pc_\sigma = \mathsf{true} \qquad wf(sh \mathbin{\underline{**}} \tau_\sigma) \qquad q' = q \setminus \Omega \cup \{\emptyset \triangleright ilREv_{\mathrm{last}(\tau)}(O) \cdot \omega\} \end{array}}{sh, q \rightarrow sh \mathbin{\underline{**}} \tau_\sigma \mathbin{\underline{**}} ilEv_{\mathrm{last}(\tau_\sigma)}(O), q'} \tag{17}$$

Select a candidate set Ω of symbolic traces representing the abstract behaviors of a method in object O with a given associated future F, and a specific local candidate trace $pc \triangleright \tau \cdot \omega$ from that set. From this trace, we select a non-empty prefix τ which we concretize with the last state σ of sh and require that the concretized path condition pc_σ holds. This rule captures interleaving, so we require that the rest ω of the trace does not introduce an internal scheduling point or diverges. This is expressed by the condition $\omega \notin \{\ldots\}$; these cases are handled by other rules below. If the extension of sh by the concrete candidate trace τ_σ is well-formed, expressed by the predicate $wf(sh \mathbin{\underline{**}} \tau_\sigma)$, the interleaving step succeeds, and the rule produces a new concrete trace $sh \mathbin{\underline{**}} \tau_\sigma \mathbin{\underline{**}} ilEv_{\mathrm{last}(\tau_\sigma)}(O)$ and a new queue q' of behaviors. The new trace ends in an interleaving event to record that τ_σ only represents a prefix of the full execution $\tau \cdot \omega$. In the new queue q',

the other possible behaviors of the current method execution in Ω are replaced by the continuation ω of the selected behavior, prefixed by the dual interleaving reaction event $ilREv_{\text{last}(\tau)}(O)$. This prefixing ensures that other behaviors of O cannot be selected for execution before this method has completed its execution.

Now consider the case where the selected behavior is a *blocking continuation* marker $blkCont(O, F, s)$, which expresses that a `get` statement is blocked while waiting for a future to be resolved.

$$\frac{\begin{array}{c} pc \triangleright \tau \cdot blkCont(O, F, s) \in \Omega \qquad \Omega \in q \qquad object(\Omega) = O \\ \tau \neq \varepsilon \qquad pc_\sigma = \mathsf{true} \qquad wf(sh \underline{**} \tau_\sigma) \qquad \text{last}(sh) = \sigma \\ q' = q \setminus \Omega \cup \{pc' \triangleright ilREv_{\text{last}(\tau)}(O) \underline{**} \omega \mid pc' \triangleright \omega \in \text{val}^{O,F}_{\text{last}(\tau)}(s)\} \end{array}}{sh, q \rightarrow sh \underline{**} \tau_\sigma \underline{**} ilEv_{\text{last}(\tau_\sigma)}(O), q'} \tag{18}$$

In contrast to rule (17), the set Ω of behaviors is here replaced by the behaviors obtained by expanding the marker to exclude local interleaving at the blocked `get` statement. Note that trace $sh \underline{**} \tau_\sigma$ ends with an event trace $blkEv_{\text{last}(\tau_\sigma)}(O, F)$.

Next consider the case when the continuation of the selected behavior has an *internal scheduling point*, as expressed by the requirement $\omega \in \{\ldots\}$:

$$\frac{\begin{array}{c} pc \triangleright \tau \cdot \omega \in \Omega \qquad \Omega \in q \qquad object(\Omega) = O \\ \tau \neq \varepsilon \qquad pc_\sigma = \mathsf{true} \qquad wf(sh \underline{**} \tau_\sigma) \qquad \text{last}(sh) = \sigma \\ \omega \in \{\varepsilon, relCont(O, \_, \_), starve(O)\} \qquad q' = q \setminus \Omega \cup \{\emptyset \triangleright \omega\} \end{array}}{sh, q \rightarrow sh \underline{**} \tau_\sigma, q'} \tag{19}$$

Interleaving events are not required, as the local semantics ensures that τ ends with a release or completion event that allows internal scheduling to happen.

Now consider the case where the selected behavior is a *release continuation* marker $relCont(O, F, s)$. In the following rule, the set Ω of behaviors is replaced by the behaviors obtained by expanding the marker at the concrete state $\text{last}(sh)$. After scheduling the trace we add an interleaving event, which allows the previous rules to concretize and decompose ω.

$$\frac{\begin{array}{c} relCont(O, F, s) \in \Omega \qquad \Omega \in q \qquad \text{last}(sh) = \sigma \\ pc_\sigma = \mathsf{true} \qquad q' = q \setminus \Omega \cup \{\emptyset \triangleright ilREv_\sigma(O) \underline{**} \omega \mid pc \triangleright \omega \in \text{val}^{O,F}_\sigma(s)\} \end{array}}{sh, q \rightarrow sh \underline{**} schEv_\sigma(O) \underline{**} ilEv_\sigma(O), q'} \tag{20}$$

Finally, consider the case where the selected behavior is *starvation*, the starving process can never be re-scheduled. This is captured by the final rule:

$$\frac{starve(O) \in \Omega \qquad \Omega \in q \qquad wf(sh \underline{**} schEv_{\text{last}(sh)}(O)) \qquad q' = q \setminus \Omega}{sh, q \rightarrow sh, q'} \tag{21}$$

In this case the concrete trace sh ends with an interleaving event trace, i.e. object O is in the middle of a sequential execution, a scheduling event for object O is technically added to sh for well-formedness checking. In addition, the set of abstract traces Ω associated to the starving method is removed from q to capture that the process never gets rescheduled.

5.3 Well-Formed Global Traces

Events in traces must obey certain ordering restrictions to ensure that only valid traces of a given program can be obtained. This is captured in the composition rules by a well-formedness predicate. Only the well-formedness of finite, concrete traces sh needs to be checked and only the information relating to events is of relevance. We use auxiliary functions $\mathit{filter}(sh, f)$ and $\mathit{filter}(sh, o)$ to filter the events related to a specific future f and to a specific object o in a finite trace sh, respectively. The output of these functions is a finite sequence η of events in which the ordering is the same as in sh. Their definition is obvious and omitted here. Well-formedness is defined inductively over the length of event sequences of a trace sh, using auxiliary predicates $\mathit{wff}(\eta, f)$ and $\mathit{wfo}(\eta, o)$:

$$\mathit{wf}(sh) \triangleq \forall o \in \mathit{obj}(sh), f \in \mathit{fut}(sh).\, \mathit{wfo}(\mathit{filter}(sh, o), o) \wedge \mathit{wff}(\mathit{filter}(sh, f), f) \quad (22)$$

Here, $\mathit{obj}(sh)$ and $\mathit{fut}(sh)$ return the set of all object and future identities found in trace sh, respectively. Thus, a global trace is well-formed if and only if the projection of its event trace on any object and any future is well-formed. We use $\mathit{ew}(\eta, o)$ and $\mathit{ew}(\eta, f)$ to return the last event in a non-empty event sequence η related to o and f, respectively. For example, $\mathit{ew}(\eta \cdot \mathit{schEv}(o), o) = \mathit{schEv}(o)$. We define the most interesting cases of $\mathit{wff}(\eta, f)$ and $\mathit{wfo}(\eta, o)$.

In a well-formed trace, a release event related to a future f can never be preceded by a completion event for f, indicating that the future is resolved. Obviously, the same holds for blocking events.

$$\mathit{wff}(\eta \frown \mathit{relEv}(o, f), f) \triangleq \mathit{wff}(\eta, f) \wedge \mathit{compEv}(o, f, \_) \notin \eta \quad (23)$$

$$\mathit{wff}(\eta \frown \mathit{blkEv}(o, f), f) \triangleq \mathit{wff}(\eta, f) \wedge \mathit{compEv}(o, f, \_) \notin \eta \quad (24)$$

We use the symbol "_" for "don't care" (implicitly universally quantified) values. To ensure cooperative scheduling (no local preemption), an interleaving event related to object o must be immediately followed by the corresponding interleaving reaction event:

$$\mathit{wfo}(\eta \frown \mathit{ilREv}(o), o) \triangleq \mathit{wff}(\eta, o) \wedge \mathit{ew}(\eta, o) = \mathit{ilEv}(o) \quad (25)$$

To prevent scheduling a different process after an interleaving event relating to an object o, an invocation reaction event or a scheduling event for o should not directly succeed an interleaving event.

$$\mathit{wfo}(\eta \frown \mathit{invREv}(\_, o, \_, \_, \_), o) \triangleq \mathit{wff}(\eta, o) \wedge (\mathit{ew}(\eta, o) \neq \mathit{ilEv}(o)) \quad (26)$$

$$\mathit{wfo}(\eta \frown \mathit{schEv}(o), o) \triangleq \mathit{wff}(\eta, o) \wedge (\mathit{ew}(\eta, o) \neq \mathit{ilEv}(o)) \quad (27)$$

The remaining cases are similar and express that, e.g., an invocation reaction happens after an invocation, scheduling only after a release or a completion, etc.

$$\varepsilon \curvearrowright \cdots \curvearrowright newEv(\mathsf{Main}, o'', \_) \curvearrowright \cdots \curvearrowright invEv(\mathsf{Main}, o'', f, \mathsf{n}, \_) \curvearrowright \cdots \curvearrowright blockEv(\mathsf{Main}, f) \curvearrowright$$
$$[l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 0] \curvearrowright ilEv(\mathsf{Main}) \curvearrowright [l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 0]$$
$$\curvearrowright$$
$$invREv(\mathsf{Main}, o'', f, \mathsf{n}, \_) \curvearrowright \cdots \curvearrowright invEv(o'', o'', f_1, \mathsf{m}, 3) \curvearrowright \cdots \curvearrowright$$
$$relEv(o'', f_1) \curvearrowright [l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 0, y' \mapsto 10, l'_1 \mapsto f_1]$$
$$\curvearrowright$$
$$invREv(o'', o'', f_1, \mathsf{m}, 3) \curvearrowright \cdots \curvearrowright$$
$$compEv(o'', f_1, 3) \curvearrowright [l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 3, y' \mapsto 10, l'_1 \mapsto f_1, x' \mapsto 3]$$
$$\curvearrowright$$
$$schEv(o'') \curvearrowright [l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 3, y' \mapsto 10, l'_1 \mapsto f_1, x' \mapsto 3] \curvearrowright ilEv(o'') \curvearrowright$$
$$[l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 3, y' \mapsto 10, l'_1 \mapsto f_1, x' \mapsto 3] \curvearrowright ilREv(o'') \curvearrowright$$
$$[l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 3, y' \mapsto 10, l'_1 \mapsto f_1, x' \mapsto 3] \curvearrowright compREv(o'', f_1, 3) \curvearrowright \cdots \curvearrowright$$
$$compEv(o'', f, 13) \curvearrowright [l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 3, y' \mapsto 13, l'_1 \mapsto f_1, x' \mapsto 3]$$
$$\curvearrowright$$
$$ilREv(\mathsf{Main}) \curvearrowright [l' \mapsto f, v' \mapsto 0, o' \mapsto o'', o''.i \mapsto 3, y' \mapsto 13, l'_1 \mapsto f_1, x' \mapsto 3] \curvearrowright$$
$$compREv(\mathsf{Main}, f, 3) \curvearrowright \cdots \curvearrowright [l' \mapsto f, v' \mapsto 3, o' \mapsto o'', o''.i \mapsto 3, \boldsymbol{y' \mapsto 13}, l'_1 \mapsto f_1, x' \mapsto 3]$$

Fig. 4. A summary of a global trace for the example.

5.4 Global Traces by Example

Figure 4 shows a possible global trace for the example in Fig. 2, by composing the local traces from Fig. 3. Note that the trace renames all declared variables and fields. The trace starts with an empty state ε in the main block, then object o is created and an asynchronous method call to method n in o is invoked. The Main object is blocked while waiting for the termination of the invoked method n (i.e. until future l is resolved). There is no local interleaving in object Main. Interleaving events are used to enable global interleaving at the blocking `get` statement. The global trace continues with the execution of method n while Main is blocked. Since the trace for method n contains a release event relating to its `await` statement, method m in object o can be selected for execution. After completion of m the remaining code of n is scheduled and once that is completed, the Main process resumes, fetches the value from future l and the program terminates when the `get` statement retrieves the value 13. Other possible global traces can be generated by varying the global interleaving at the `get` statement in the Main block and the `await` statement in method n.

6 Calculus

The main point of a modular, denotational semantics for ABS is to drive the development of program logics for deductive reasoning. Although this will largely be the topic of future work, we sketch some opportunities. Starting points are (i) the dynamic logic for *sequential* Java implemented in KeY [3], (ii) the dynamic logic for the ABS version of KeY [16] that permits *object-local* reasoning about class invariants, and (iii) a dynamic logic for a sequential language that uses symbolic trace formulas to specify program behavior [17]. Trace formulas are the

syntactic counterpart to symbolic traces. All three logics implement a *symbolic interpreter* for the sequential language fragments in their calculus. Therefore, they are an excellent match for the semantics developed here—in fact, program logics were the motivation for the work presented in this paper. We will merge these program logics into a single one that is sufficient for *invariant* reasoning on *local objects* with *symbolic traces*. The logic outlined here is relatively weak, because global behavior or pre- and postconditions (specifically, return values) are not addressed, but it is designed with suitable extensions in mind.

6.1 Symbolic Trace Formulas

Our semantics uses symbolic traces to specify the behaviors of local computations of ABS programs. It is, therefore, natural to have a syntactic representation of them in the logic. Symbolic trace formulas are due to Nakata & Uustalu [14], and were intended for an abstract Hoare calculus with co-inductive reasoning about non-terminating programs. Trace formulas were generalized to dynamic logic over trace modality formulas in [17]. For our purposes, it is sufficient to leave trace formulas completely abstract, i.e., a *trace formula* is an expression Θ that describes a possibly infinite set of (concrete) traces. There is a semantic evaluation function such that $\mathrm{val}_\tau^{O,F}(\Theta)$ is true iff τ is one of the traces on O, F described by Θ. We give an informal example of a typical trace formula:

$$\mathit{invREv}(\_, \texttt{this}, f, m, v) \ll \mathit{invEv}(\texttt{this}, \_, \_, n, w) \mathbin{**} \lceil \phi(v, w) \rceil \ll \mathit{compEv}(\texttt{this}, f, m)$$

Here, $\ll$ represents a "happens after" relation (i.e. an arbitrary, finite trace between the events), $**$ is the syntactic equivalent of $\underline{**}$, and $\lceil \phi \rceil$ denotes the occurrence of a state in which ϕ holds. The trace formula above might be paraphrased as: "whenever the current object '`this`' completes method m, then during the execution of m there was a call to n such that the arguments v of m and w of n were in relation ϕ." This is a typical example of a class invariant that can be succinctly expressed via symbolic trace formulas.

Trace modality formulas Ψ are defined on top of trace formulas by taking them as atomic building blocks that are syntactically closed with respect to the usual propositional/first-order connectives and the following modalities:

1. If s is an ABS statement and Ψ a trace modality formula, then $[\![s]\!]\Psi$ is a trace modality formula.
2. If $\{u\}$ is a trace update and Ψ a trace modality formula, then $\{u\}\Psi$ is a trace modality formula.

Trace updates [17] are expressions $\{\ell := exp\}$ or $\{ev(\overline{e})\}$ recording state change effected by assignments (with semantics $\langle\sigma\rangle \rightsquigarrow \sigma[\ell \mapsto \mathrm{val}_\sigma(exp)]$) or the occurrence of communication events. Let $\mathcal{U}$ denote a finite sequence of trace updates.

The semantics $\mathrm{val}_\tau([\![s]\!]\Psi)$ of a trace modality formula $[\![s]\!]\Psi$ and a trace τ is formally defined as: if τ is finite, $O \in \mathcal{O}$, $F \in \mathcal{F}$, and $\tau' \in \mathrm{val}^{O,F}_{\mathrm{last}(\tau)}(s)$, then $\tau \underline{**} \tau'$, if well-formed, is in $\mathrm{val}_\tau^{O,F}(\Psi)$. In words, any valid trace of s that extends

τ must be a trace of Ψ. If τ is infinite, s is never reached, and τ must be a trace of Ψ. The semantics of $\{u\}\Psi$ is similar, by first extending τ according to the trace update u.

6.2 Selected Reasoning Rules

We define a sequent calculus where antecedents and succedents range over multi-sets of trace modality formulas. For trace modality formulas Γ and $[\![s]\!]\Psi$, and trace updates $\mathcal{U}$, the sequent $\Gamma \Rightarrow \mathcal{U}[\![s]\!]\Psi$ expresses: if the execution of s on `this` with future `destiny` begins in the last state of a finite trace τ described by Γ and after applying $\mathcal{U}$, then Ψ must contain any trace generated by the execution of s and the effect of the updates $\mathcal{U}$ that extends τ. For an infinite trace τ in Γ, s is not executed, but Ψ must contain τ.

We now discuss four proof rules in more detail. In the rule for *assignments*,

$$\mathsf{assign}\ \frac{\Gamma \Rightarrow \mathcal{U}\{\ell := e\}[\![r]\!]\Psi}{\Gamma \Rightarrow \mathcal{U}[\![\ell = e;\, r]\!]\Psi}$$

ℓ is a program variable and e a pure (side effect-free) expression. This rule rewrites the formula by moving the assignment from the program into an update to capture the state change, here $\{\ell := e\}$. Symbolic execution continues with the remaining program r. Updates can be viewed as explicit substitutions accumulated in front of the modality during symbolic program execution. Once the program has been completely executed and the modality is empty, the accumulated updates are applied to the formula after the modality, resulting in an update- and program-free trace modality formula. In the rule for *asynchronous method calls*

$$\mathsf{asyncCall}\ \frac{\Gamma, \mathit{isFresh}(f) \Rightarrow \mathcal{U}\{\mathit{invEv}(O, \mathtt{this}, f, m, \overline{e}')\}\{\ell := f\}[\![r]\!]\Psi}{\Gamma \Rightarrow \mathcal{U}[\![\ell = e!m(\overline{e}');\, r]\!]\Psi}$$

the premise introduces a constant f representing the future associated with this method invocation. The left side of the implication ensures that f is fresh. The right side adds two trace updates, an invocation event generated by this call and a binding of ℓ to the fresh future f. In the rule for the *await* statement

$$\mathsf{await}\ \frac{\begin{array}{c}\Gamma, \mathit{class}(\mathtt{this}) = C \Rightarrow \mathcal{U}\, I_C \\ \Gamma, \mathcal{U}\,\mathcal{U}_a\, I_C \Rightarrow \mathcal{U}\,\mathcal{U}_a\,\{\mathit{compREv}(\mathtt{this}, \ell, v_0)\}[\![r]\!]\Psi\end{array}}{\Gamma \Rightarrow \mathcal{U}[\![\mathtt{await}\ \ell?;\, r]\!]\Psi}$$

I_C denotes a trace modality formula that serves as the invariant of class C. This rule has two premises: the first expresses that I_C should hold at the process release point and the second expresses the situation where future ℓ has been resolved. This is captured by the completion reaction event. Update $\mathcal{U}_a$ represents the unknown trace updates from other processes on the same object. This is achieved by initializing any location that might be changed by another process

with a fresh constant. Since class invariant I_C is guaranteed by the latest released process, I_C is true by assumption. The rule for the *get* statement is

$$\textsf{get}\ \frac{\Gamma, \mathit{isFresh}(v_0) \Rightarrow \mathcal{U}\{\mathit{compREv}(\texttt{this}, \ell', v_0)\}\{\ell := v_0\}[\![r]\!]\Psi}{\Gamma \Rightarrow \mathcal{U}[\![\ell = \ell'.\texttt{get}; r]\!]\Psi}$$

For partial correctness, we can assume that future ℓ' has been resolved. The right side of the implication adds two trace updates, a completion reaction event for ℓ' with some value v_0 and a binding of ℓ to v_0. Note that v_0 is a fresh variable because ℓ' might be resolved on a different object, which makes the value of v_0 invisible. Any information about the value of ℓ must be put into the class invariant. However, support from pre- and postcondition reasoning in future work may provide more information about the return value v_0.

7 Related Work

This paper is motivated by our aim to devise compositional proof systems to verify protocol-like behaviors for asynchronously communicating objects. The general field of trace semantics is too vast to cover here. For languages with ABS-like features, Din et al. [16,18,19] introduced 4-event trace semantics for asynchronous method calls and shared futures, which, together with the trace modality formulas of Nakata et al. [14,17], underlies our work. Recent work [20] on similar communication structures for ASP/ProActive, using parametrized labelled transition systems with queues, models interaction with futures in a very detailed, operational way. In contrast, our work with traces allows futures to be abstracted into communication events and well-formedness conditions.

Brookes' action traces [10] bear some similarity to our work. He aims at denotational semantics using collecting semantics, explicitly represents divergence, and synchronizes communication using events. Action traces have been used as a semantics for concurrent separation logic [21], where scheduling is based on access to shared resources with associated invariants (so-called "mutex fairmerge"). In contrast, we use conditioned traces and continuations, cover procedure calls by *abstract* traces, and extend the use of dual events from communication to different scheduling situations, resulting in a compositional denotational semantics for asynchronous method calls and cooperative concurrency.

8 Conclusion and Future Work

We presented a denotational semantics for an OO concurrent language with cooperative scheduling that is streamlined for the development of program logics with trace formulas. The main advantages of the semantics are its compositionality and the separation of local and global computations. Technical innovations include abstract, conditioned traces permitting symbolic evaluation as well as event pairs to keep track of schedulability. We sketched a simple program logic with trace formulas that is sufficient for local invariant reasoning. In future work

we will extend it to a calculus that allows to reason about global properties, including liveness, and that supports method-local specification with contracts.

Acknowledgement. We are grateful to Dave Sands for useful hints and feedback and to Georges P. for inspiring our use of constraints.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). (583)
2. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle (1976)
3. Beckert, B., Klebanov, V., Weiß, B.: Dynamic logic for Java. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P., Ulbrich, M. (eds.) Deductive Software Verification–The KeY Book: From Theory to Practice. LNCS, vol. 10001, pp. 49–106. Springer, Heidelberg (2016). doi:10.1007/978-3-319-49812-6
4. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebraic Program. **60–61**, 17–139 (2004)
5. Drossopoulou, S., Eisenbach, S.: Describing the semantics of Java and proving type soundness. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. LNCS, vol. 1523, pp. 41–82. Springer, Heidelberg (1999)
6. Krebbers, R., Wiedijk, F.: A typed C11 semantics for interactive theorem proving. In: Conference on Certified Programs and Proofs, 15–27. ACM (2015)
7. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), vol. 6957, pp. 142–164. Springer, Berlin (2011). doi:10.1007/978-3-642-25271-6_8
8. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) 19th International Conference on Computer Aided Verification, CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Berlin (2007). doi:10.1007/978-3-540-73368-3_21
9. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17511-4_20
10. Brookes, S.: Traces, pomsets, fairness and full abstraction for communicating processes. In: Brim, L., Křetínský, M., Kučera, A., Jančar, P. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 466–482. Springer, Heidelberg (2002). doi:10.1007/3-540-45694-5_31
11. Meyer, B.: Applying "design by contract". IEEE Comput. **25**(10), 40–51 (1992)
12. Nielson, F., Nielson, H.R., Hankin, C.L.: Principles of Program Analysis. Springer, Heidelberg (1999). doi:10.1007/978-3-662-03811-6
13. Hentschel, M., Hähnle, R., Bubel, R.: Visualizing unbounded symbolic execution. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 82–98. Springer, Cham (2014). doi:10.1007/978-3-319-09099-3_7
14. Nakata, K., Uustalu, T.: A Hoare logic for the coinductive trace-based big-step semantics of While. Log. Methods Comput. Sci. **11**(1), 1–32 (2015)
15. Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71316-6_22

16. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS, vol. 9195, pp. 517–526. Springer, Cham (2015). doi:10.1007/978-3-319-21401-6_35
17. Bubel, R., Din, C.C., Hähnle, R., Nakata, K.: A dynamic logic with traces and coinduction. In: Nivelle, H. (ed.) TABLEAUX 2015. LNCS, vol. 9323, pp. 307–322. Springer, Cham (2015). doi:10.1007/978-3-319-24312-2_21
18. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: component reasoning for concurrent objects. J. Logic Algebraic Program. **81**(3), 227–256 (2012)
19. Din, C.C., Owe, O.: Compositional reasoning about active objects with shared futures. Formal Asp. Comput. **27**(3), 551–572 (2015)
20. Ameur-Boulifa, R., Henrio, L., Kulankhina, O., Madelaine, E., Savu, A.: Behavioural semantics for asynchronous components. J. Logical Algebraic Methods Program. **89**, 1–40 (2017)
21. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1–3), 227–270 (2007)

http://www.springer.com/978-3-319-66901-4

Automated Reasoning with Analytic Tableaux and Related Methods
26th International Conference, TABLEAUX 2017, Brasília, Brazil, September 25–28, 2017, Proceedings
Schmidt, R.A.; Nalon, C. (Eds.)
2017, XII, 381 p. 75 illus., Softcover
ISBN: 978-3-319-66901-4